虚拟机面试一般包含以下几个知识点：

JVM内存划分

JVM垃圾回收，垃圾回收可从几个点出发：

什么对象需要回收

什么时候回收

怎么回收

垃圾回收算法

垃圾收集器

如何使用工具观察和解决虚拟机问题

参数调优

类加载机制

加载过程

双亲委派原理

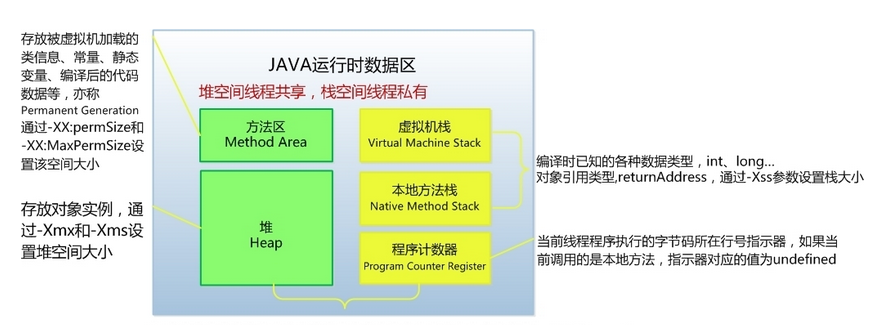
执行引擎

动态委派和静态委派。这里涉及到java多态的概念。

# 一、JVM内存区域与内存溢出

# JVM内存划分

<https://blog.csdn.net/csdn___lyy/article/details/79450860>



## 1.1 程序计数器 （线程私有）

1.占有较小的内存空间，可以看作是当前所执行字节码的行号指示器；

2. 为了线程切换后能恢复到正确的执行位置，**每条线程**都需要一个独立的程序计数器，各线程之间的计数器互不影响，独立存储。

3.当线程执行java方法时，记录的是正在执行的jvm字节码指令的地址。若执行的为Native方法，计数器值为空即undefined；

4.唯一一个在jvm中没有规定OutOfMemoryError异常的区域。

## 1.2 java虚拟机栈（线程私有）

 1.**线程私有**，生命周期与线程相同。

 2.描述的是Java方法执行的内存模型：每一个方法执行的同时都会创建一个栈帧（Stack Frame），用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法的执行就对应着栈帧在虚拟机栈中的入栈，出栈过程。

**局部变量表**：存放编译期可知的各种基本数据类型、对象引用类型和returnAddress类型（指向一条字节码指令的地址：函数返回地址）。局部变量表所需的内存空间在编译期确定，当进入一个方法时，方法在栈帧中所需要分配的局部变量控件是完全确定的，不可动态改变大小。

## 1.3.本地方法栈（线程私有）

虚拟机栈为虚拟机执行 Java 方法（字节码）服务。

本地方法栈（Native Method Stacks）为虚拟机使用到的 Native 方法服务。

HotSpot虚拟机中把本地方法栈和虚拟机栈合二为一。

## 1.4java堆（线程共享）

1.JVM中所管理内存中的最大的一块。在虚拟机启动时被创建。

2.唯一的目的是存放**对象实例**，几乎所有的对象实例和数组都是在这里分配内存。（JVM规范中说的是所有的，但是随着JIT便编译器的发展和逃逸技术分析的成熟，一些实例可以不在这个区域分配内存） 。

3.堆是垃圾收集管理的主要区域，所以也会被称为”GC堆。

## 1.5方法区（线程共享）

用于存储虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

这区域的内存回收主要是针对常量池的回收和对类的卸载。

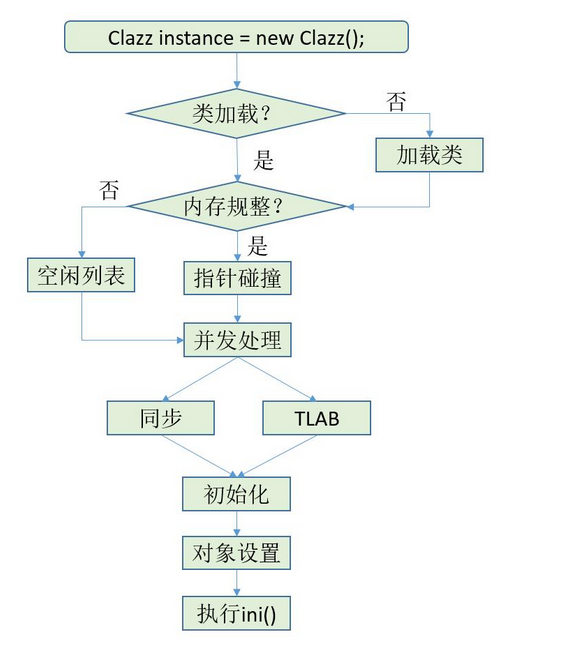
方法区也可被垃圾收集，例如：在HotSpot中将GC分代收集扩展至方法区。避免了还要再为方法区编写GC算法。

## 1.6运行时常量池

1.运行时常量池是方法区的一部分，用于存放编译期生成的各种字面量和符号引用，这部分的内容将在类加载后进入方法区的运行时常量池中。

2.运行时常量池具备动态性，运行期间也可以将新的常量放入池中，例如String类的intern()方法。

# Java对象在Java虚拟机中的创建过程（创建对象怎么分配内存）



## 1. 相应类加载检查过程

虚拟机遇到一条new指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

## 2. ****为对象分配内存****

 对象所需内存的大小在类加载完成后便完全确定（JVM可以通过普通Java对象的类元数据信息确定对象大小），为对象分配内存相当于把一块确定大小的内存从Java堆里划分出来。

2.1内存的分配方式

**（I）指针碰撞**

      如果Java堆是绝对规整的：一边是用过的内存，一边是空闲的内存，中间一个指针作为边界指示器； 分配内存只需向空闲那边移动指针，这种分配方式称为"指针碰撞"；

**（II）空闲列表**

      如果Java堆不是规整的：用过的和空闲的内存相互交错；需要维护一个列表，记录哪些内存可用；分配内存时查表找到一个足够大的内存，并更新列表，这种分配方式称为"空闲列表"；

Java堆是否规整由JVM采用的垃圾收集器是否带有压缩功能决定的；所以，使用Serial、ParNew等带Compact过程的收集器时，JVM采用指针碰撞方式分配内存；而使用CMS这种基于标记-清除（Mark-Sweep）算法的收集器时，采用空闲列表方式。

2.2线程安全问题

      并发时，上面两种方式分配内存的操作都不是线程安全的，有两种解决方案：

**（I）同步处理**

      对分配内存的动作进行同步处理：

      JVM采用CAS（Compare and Swap）机制加上失败重试的方式，保证更新操作的原子性；

      CAS：有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做；

**（II）本地线程分配缓冲区**

      把分配内存的动作按照线程划分在不同的空间中进行：

      即每个线程在Java堆预先分配一小块内存，称为本地线程分配缓冲区（Thread Local Allocation Buffer,TLAB）；哪个线程需要分配内存就从哪个线程的TLAB上分配；只有TLAB用完需要分配新的TLAB时，才需要同步处理；

JVM通过"-XX：+/-UseTLAB"指定是否使用TLAB。

## 3.对象内存初始化为零

      对象内存初始化为零，但不包括对象头； 如果使用TLAB，提前至分配TLAB时；

      这保证了程序中对象（及实例变量）不显式初始赋零值，程序也能访问到零值；

## 4. 对对象进行必要的设置

接下来要对对象进行必要的设置。主要设置对象头信息，包括类元数据引用、对象的哈希码、对象的GC分代年龄等。

## 5.执行对象实例方法<init>

      该方法把对象（实例变量）按照程序中定义的初始赋值进行初始化。

      通常，经过上面5步对象才完全new出来。

# 虚拟机中对象的内存布局

## ****对象内存布局****

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头(Header)、实例数据(Instance Data)和对齐填充(Padding)。

### ****1.对象头****

HotSpot虚拟机的对象头包括两部分信息：运行时数据和类型指针。

### ****1.1运行时数据（Mark Word）****

用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。

对于对象头长度而言 :

32bit虚拟机一定是32bit+32bit，即8字节；64bit虚拟机若没有开启了压缩指针，是64bit+64bit，即16字节，若开启了压缩指针，是64bit+32bit，即12字节（不是8bit的倍数）。-XX:+UseCompressedOops：开启压缩指针。

例如在32位的HotSpot虚拟机 中对象未被锁定的状态下，Mark Word的32个Bits空间中的25Bits用于存储对象哈希码（HashCode），4Bits用于存储对象分代年龄，2Bits用于存储锁标志 位，1Bit固定为0，在其他状态（轻量级锁定、重量级锁定、GC标记、可偏向）下对象的存储内容如下表所示。





### ****1.2类型指针****

即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通Java对象的元数据信息确定Java对象的大小，但是从数组的元数据中无法确定数组的大小。   
(并不是所有的虚拟机实现都必须在对象数据上保留类型指针，换句话说，查找对象的元数据并不一定要经过对象本身，可参考对象的访问定位)。

### ****2.实例数据****

实例数据部分是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。无论是从父类中继承下来的，还是在子类中定义的，都需要记录下来。HotSpot虚拟机默认的分配策略为longs/doubles、ints、shorts/chars、bytes/booleans、oop，从分配策略中可以看出，相同宽度的字段总是分配到一起。

### ****3.对齐填充****

对齐填充不是必然存在的，没有特别的含义，它仅起到占位符的作用。

HotSpot虚拟机要求对象的起始地址必须是8字节的整数倍，也就是对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的倍数（1倍或者2倍），因此，当对象实例数据部分没有对齐的时候，就需要通过对齐填充来补全。

### ****4.对象的访问定位****

4.1句柄

java栈本地变量表中 reference指向句柄，句柄指向实例数据和类型数据;

reference存放的是稳定的句柄地址;

对象移动只会改变句柄中的实例数据指针，不改变reference.

4.2直接指针访问

速度快

节省开销

# 内存泄漏了解多少，想办法写程序把堆内存搞垮，把栈内存呢？方法区呢？

**1.程序计数器**：

唯一一块不报OutOfMemoryError异常的内存区域。

**2.Java虚拟机栈**：

StackOverflowError：线程请求的栈深度大于虚拟机所允许的最大深度。

OutOfMemoryErro：如果虚拟机栈可以动态**扩展**，当拓展时无法申请到足够的内存空间。

**3.本地方法栈**：

StackOverflowError：与虚拟机栈相似

OutOfMemoryError：与虚拟机栈相似

**4.Java堆**:

OutOfMemoryError：堆上没有内存完成实例的分配

5.**方法区：**

**类信息、常量、静态变量、即时编译器编译后的代码**等数据。

OutOfMemoryError：当方法区无法满足内存分配需求时

**6.运行时常量池**：

用于存放**编译器生成**的各种**字面量和符号引用，**是方法区的一部分。

OutOfMemoryError： 无法申请内存

# 对jvm中堆和栈的理解

Java 把内存划分成两种：一种是栈内存，另一种是堆内存。栈是java运行的单位 ，堆是存储的单位。

栈区:

1. **每个线程包含一个栈区**，栈中只保存**基础数据类型和自定义对象的引用(不是对象)**，对象都存放在堆区中；
2. 每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问；
3. 栈分为3个部分：基本类型变量区、执行环境上下文、操作指令区(存放操作指令)。
4. 如果栈内存没有可用的空间存储方法调用和局部变量，JVM会抛出java.lang.StackOverFlowError。
5. 当代码块中定义一个变量时，java会自动分配内存。当超过变量的作用域后，java会自动释放掉这块内存。

堆区:

1. 存储的全部是对象实例，每个对象都包含一个与之对应的class的信息(class信息存放在方法区)。在栈中定义一个特殊的变量，让栈中的这个变量的取值等于数组或对象在堆内存中的首地址。
2. **jvm堆区(heap)被所有线程共享**，堆中不存放基本类型和对象引用，只存放对象本身，几乎所有的**对象实例和数组**都在堆中分配。
3. 在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理，但不是立马释放堆内存。

4.而如果是堆内存没有可用的空间存储生成的对象，JVM会抛出java.lang.OutOfMemoryError。

# Student s= new Student(),在内存中做了那些事情

这条语句的执行，即是Java对象在Java虚拟机中的创建过程。

载入Student.class文件进内存（方法区）

在栈内存为s开辟空间

在堆内存为学生对象开辟空间

对学生对象的成员变量进行默认初始化

对学生对象的成员变量进行显示初始化

通过构造方法对学生对象的成员变量赋值

学生对象初始化完成，把对象地址赋值给s变量

# JVM创建对象怎么保证线程安全

<https://blog.csdn.net/sunny_aaadolly/article/details/78924799>

虚拟机在创建对象的时候采用两种方法：

1. 指针碰撞
2. 空闲列表

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：CAS+失败重试和TLAB。

## 1.CAS+失败重试

CAS是乐观锁的一种实现方式，通过比对原值和旧的预期值来确定是否要将原值更改为新值，如果通过CAS来实现线程安全，那么需要三个因子，首先是在主内存中的一个原值，然后第二个是这个原值在各个线程中的副本，再接下来就是新值。

1.1 指针碰撞的方式

采用指针碰撞的方式进行对象的内存分配，那么这个原值就是当前指针的位置。假设说现在是初始化状态，那么指针的位置就是0，也就是原值就是0，这个0会在主内存中存放着。假设现在有两个线程A和B，每个线程都在执行着创建对象的操作，这两个线程中保存着原值的副本，此时的原值是0，副本也是0，新值就是需要的内存量，假设说对象A需要5的内存，对象B需要7的内存。

现在两个线程开始创建对象，我们采用CAS+失败重试的方法来确保线程安全。

首先A线程开始创建对象，此时触发乐观锁的机制，A读取到了目前的内存情况，也就是指针的初始位置0，因为是乐观锁，所以在没有提交的时候并不会触发冲突检查。这个时候时间切换到线程B，线程B也开始创建对象，同样读取到了目前的内存情况，同样的指针位置是0，因为没有提交，同样不会触发冲突检查。但是线程的工作是CPU轮流安排时间片进行的，同一时间只会有一条线程执行任务，这个时候又切换到线程A，A开始提交，提交的时候触发了冲突检查，对原值以及旧的预期值进行比对，原值是0，预期原值也就是线程里面的原值副本，此时线程A的旧的预期值也是0，比对通过，将新值赋给原值，新值是5，那么指针就会移动5个单位，此时指针的位置就是在5，同时将线程A的预期值更改为5。

现在时间再次切换到线程B，线程B开始提交，触发乐观锁机制，进行冲突检查，此时的原值是5，线程B的旧的预期值是0，比对不通过，此时虚拟机什么都不做，只是把线程B的旧的预期值更新为5，然后触发失败重试，线程B会再次尝试提交，再次出发冲突检查，那么这个时候检查就通过了，通过之后将新值赋给原值，需要注意的地方是，这个赋值并不是把5改变成7，而是将指针移动7个单位，移动到12，然后再把线程B的旧的预期值更新为12，以此类推。

1.2空闲列表的方式

同样的A、B两个线程都需要创建对象分配空间，A需要5，B需要7。

首先是A线程，此时的原值是可用的内存空间0-100，A和B的旧的预期值也都是0-100，这个时候A先去创建对象，检查通过后，虚拟机分配了7-12的内存给了A线程的对象，同时将原值以及A线程的旧的预期值更改为0-7，12-100。这个时候B再去提交创建对象的时候，旧的预期值是0-100，与0-7，12-100比对不通过，将预期值更新为0-7，12-100，然后触发失败重试，再次提交的时候比对通过，提交成功，更新原值和线程B的预期值，以此类推。

## 2. TLAB

TLAB，本地线程分配缓冲。每个线程都会在Eden空间申请到一个TLAB，大小占Eden空间的1%，当然申请这个空间的过程是线程同步的，这个同步的实现也是依赖于CAS+失败重试的方式。可以把TLAB想象成一个对象，占用内存大小就是Eden空间的1%即可。

当这个线程需要创建对象的时候，直接在TLAB里面创建就行了，这样就避免因并发而导致的线程安全问题。

当然，有这样的一种情况，现在这个线程需要创建一个对象，但是当前的TLAB的空间不足了，怎么办，它会再向Eden空间去申请一个TLAB，申请的过程是线程同步的，它会把这个对象放到新的TLAB中，也就是说一个线程并不是只有一个TLAB。

那如果这个对象特别大，哪怕是一个新的TLAB也放不下呢？直到这个时候，线程才会去把对象直接创建在Eden空间，再次采用CAS+失败重试的方法去保证线程同步。

也就是说，采用这种方式，线程会向Eden空间申请线程私有的TLAB来创建对象，确保线程安全，除非说现在的TLAB不够用了，再去申请新的TLAB的时候才会同步锁定，或者说是对象特别大，一个全新的TLAB空间都装不下了，必须去Eden空间创建，才会同步锁定。

但一般来说，创建的对象都是特别小的，也都是会迅速销毁的，所以这种方式从效率上来讲还是比较高的。

# JAVA中字符串内存位置浅析

基本类型的变量数据和对象的引用都是放在栈里面的，对象本身放在堆里面，显式的String常量放在常量池，String对象放在堆中。

### String内存位置说明

1.显式的String常量



第一句代码执行后就在常量池中创建了一个值为holten的String对象；

第二句执行时，因为常量池中存在holten所以就不再创建新的String对象了。

此时该字符串的引用在虚拟机栈里面。

2.String对象



Class被加载时就在常量池中创建了一个值为holtenObj的String对象，第一句执行时会在堆里创建new String("holtenObj")对象；

第二句执行时，因为常量池中存在holtenObj所以就不再创建新的String对象了，直接在堆里创建new String("holtenObj")对象。



# 二、JVM垃圾收集

## 判断对象死亡

### 1.1 引用计数器法

   算法思想：为对象添加一个引用计数器，每当有一个地方引用该对象时，则该引用计数器值加1,；当引用失效时，则该引用计数器值减1；最后，计数器为0的对象就是不可能再被使用的，也即所谓的“死去”的对象。

Java中并没有使用这种算法进行GC，最主要的原因是很难解决对象之间的相互循环引用的问题。

### 1.2可达性分析法

    Java中是使用可达性分析算法判断对象是否存活的。

  算法思想：选择一系列名为“GC Roots”的对象为起始点，从这些GC Roots节点开始向下搜索，搜索走过的路径称之为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链时（即不可达，两者之间无通路），则认为该对象为不可用的。

1.2.1 在java语言中，可作为GC Roots的对象包括以下几种：

a、虚拟机栈（栈帧中的本地变量表）中的引用的对象；

    b、方法区中的常量引用的对象；

    c、方法区中的类静态属性引用的对象；

d、本地方法栈中JNI（Native方法）的引用的对象。

## 引用的分类

a、强引用（Strong Reference）

          程序代码中普遍存在的，比如Object obj = new Object()，只要存在强引用，GC收集器永远不会回收被引用的对象。

b、软引用（Soft Reference）

描述一些**有用但并非必须**的对象。

          非必须的对象，是否回收，要看当前内存情况，如果紧张，则进行回收，否则不回收。对于软引用关联的对象在系统将要**发生内存溢出异常之前**，将会把这些对象**列进回收范围之中进行第二次回收。**当程序抛出内存溢出异常时，肯定不存在这种类型的对象。

    c、弱引用（Weak Reference）

          比软引用还要弱一些。被弱引用关联的对象只能生存到下一次GC发生之前。无论当前内存是否足够，都会回收掉只被弱引用关联的对象。

    d、虚引用（Phantom Reference）

          幽灵引用或者幻影引用，一个对象是否有虚引用的存在不会影响到其生存时间，无法通过虚引用获取对象实例。为一个对象设置虚引用关联的唯一目的是希望能在这个对象被回收时受到一个系统通知。

## 判断一个对象真正的死亡

要宣告一个对象真正的死亡，至少要经历两次标记的过程。

1、  如果对象在进行可达性分析后发现没有与GC Root相连接的引用链，将会被第一次标记，并且进行一次筛选，筛选条件是对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法或finalize()方法已经被虚拟机调用过，虚拟机将这两种情况称为没有必要执行。

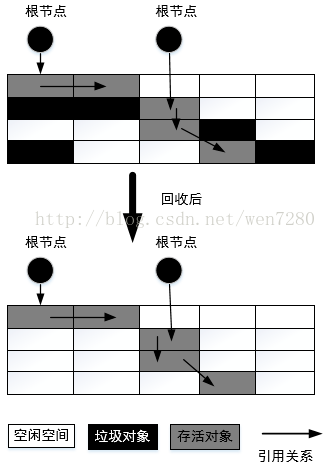
        2、  如果对象被判定为有必要执行finalize()方法，那么该对象将会被放置到低优先级的Finalize线程去执行它，但不承若会等待它运行结束，GC将对F\_Queue中的对象进行第二次小规模的标记，如果对象要在finalize()方法中成功解救自己，只要重新与引用链上的任何一个对象建立关联即可，如果对象这时候还没有逃脱那基本上它就真的被回收了。

## 垃圾收集算法

### 4.1 标记-清除算法

 最基础的收集算法，分为“标记”和“清除”两个阶段，首先标记所有需要回收的对象，然后统一回收所有标记对象 。 后续算法都是基于此思想并改进 。

主要缺点：一、效率低，二、空间浪费，标记清除之后会产生大量不连续的内存碎片，当需要分配较大对象时，不得不提前触发另一次垃圾回收。



### 4.2复制算法

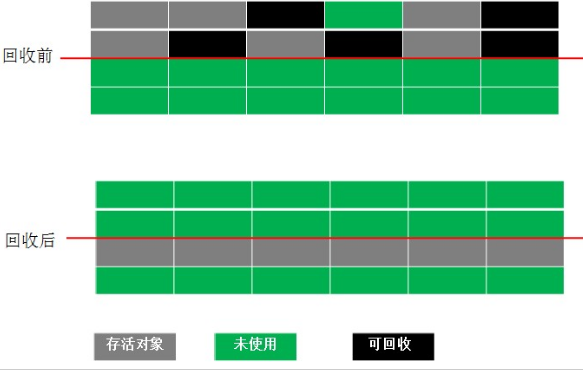
为解决效率问题，提出复制的算法。

       将内存一分为相等的两块，每次只使用一块。当这一块内存用光，就将还存活的对象复制到另外一块，然后再把已经使用过的空间一次清理掉。

       优点：每次只针对其中一块进行内存回收，内存分配也不用考虑内存碎片等复杂情况，只要一点堆顶指针，按顺序分配内存即可。实现简单，运行高效。

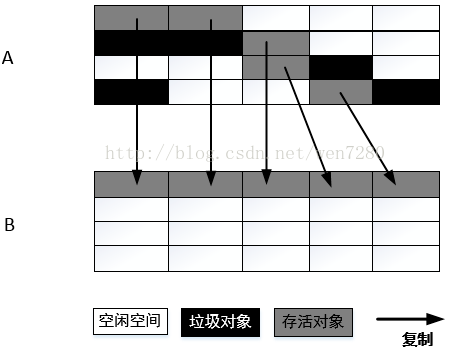
       缺点：内存缩小为原来的一半。

一般用于新生代。一般是将内存分为一块较大的**Eden空间**和两块较小的**Survivor**空间。HotSpot虚拟机默认比例是**8:1**,。每次使用Eden和一块Survivor，当回收时将这两块内存中还存活的对象复制到未使用的Survivor，然后清理掉Eden和刚擦使用过的Survivor的空间。如果复制过程内存不够使用则向老年代分配担保。



### 4.3标记-整理算法

根据老年代的特点，提出“标记-整理”算法。标记的过程与“标记-清除”的算法一样，但是后序步骤不是知己对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。



### 4.4分代收集算法

根据对象的生存周期将内存划分为**新生代和老年代**，根据年代的特点采用最适当的收集算法。

**新生代：复制**

**老年代：标记-清除或标记-整理。**

## 垃圾收集器

没有最好的收集器，更没有万能的收集；选择的只能是适合具体应用场景的收集器。

在谈垃圾收集器的上下文语境中，

并行：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

并发：指用户线程和垃圾收集线程同时执行，用户程序在继续运行，而垃圾收程序运行在另一个CPU上。

### 5.1 ****Serial收集器****

Serial（串行）垃圾收集器是最基本、发展历史最悠久的收集器。

1、特点

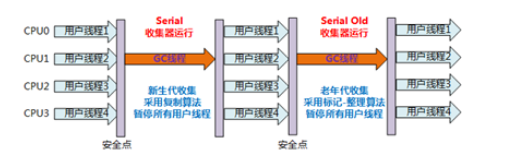
针对新生代；

采用复制算法；

单线程收集；

进行垃圾收集时，必须暂停所有工作线程，直到完成，即会"Stop The World"。

**Serial/Serial Old组合收集器运行示意图如下：**



2.  应用场景

依然是HotSpot在Client模式下默认的新生代收集器。

3、设置参数

     "-XX:+UseSerialGC"：添加该参数来显式的使用串行垃圾收集器。

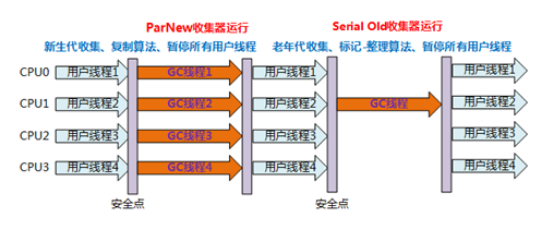
### 5.2 ParNew收集器

ParNew垃圾收集器是**Serial收集器的多线程版本**。

1.特点

 除了多线程外，其余的行为、特点和Serial收集器一样。

**ParNew/Serial Old组合收集器运行示意图如下：**



2.应用场景

在Server模式下，ParNew收集器是一个非常重要的收集器，因为除Serial外，目前只有它能与CMS收集器配合工作。

在单个CPU环境中，不会比Serail收集器有更好的效果，因为存在线程交互开销。

3、设置参数

**"-XX:+UseConcMarkSweepGC"**：指定使用CMS后，会默认使用ParNew作为新生代收集器；

**"-XX:+UseParNewGC"**：强制指定使用ParNew；

**"-XX:ParallelGCThreads"**：指定垃圾收集的线程数量，ParNew默认开启的收集线程与CPU的数量相同。

### 5.3 ****Parallel Scavenge收集器****

 Parallel Scavenge垃圾收集器因为与吞吐量关系密切，也称为**吞吐量收集器（Throughput Collector）**。

1.特点

1.1有一些特点与ParNew收集器相似：新生代收集器；采用复制算法；多线程收集。

1.2主要特点：它的关注点与其他收集器不同

      CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间；

      而Parallel Scavenge收集器的目标则是达一个可控制的吞吐量（Throughput）。

2.应用场景

高吞吐量为目标，即减少垃圾收集时间，让用户代码获得更长的运行时间；

当应用程序运行在具有多个CPU上，对暂停时间没有特别高的要求时，即程序主要在后台进行计算，而不需要与用户进行太多交互。例如，那些执行批量处理、订单处理、工资支付、科学计算的应用程序。

3.设置参数

**（A）"-XX:MaxGCPauseMillis"**

      控制最大垃圾收集停顿时间，大于0的毫秒数；

      MaxGCPauseMillis设置得稍小，停顿时间可能会缩短，但也可能会使得吞吐量下降；

      因为可能导致垃圾收集发生得更频繁；

**（B）"-XX:GCTimeRatio"**

      设置垃圾收集时间占总时间的比率，0<n<100的整数；

      GCTimeRatio相当于设置吞吐量大小；

**（C）"-XX:+UseAdptiveSizePolicy"**

开启这个参数后，就不用手工指定一些细节参数，JVM会根据当前系统运行情况收集性能监控信息，动态调整这些参数，以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为**GC自适应的调节策略（GC Ergonomiscs）**。

### 5.4 ****Serial Old收集器****

Serial Old是 **Serial收集器的老年代版本**。

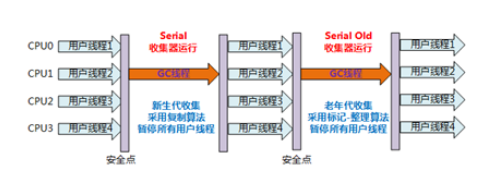
1、特点

      针对老年代；

      采用"标记-整理"算法；

      单线程收集；

 Serial/Serial Old收集器运行示意图如下：



2、应用场景

      主要用于Client模式；

而在Server模式有两大用途：

      （A）在JDK1.5及之前，与Parallel Scavenge收集器搭配使用（JDK1.6有Parallel Old收集器可搭配）；

      （B）作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用；

### 5.5 ****Parallel Old收集器****

Parallel Old垃圾收集器是Parallel Scavenge收集器的老年代版本；JDK1.6中才开始提供。

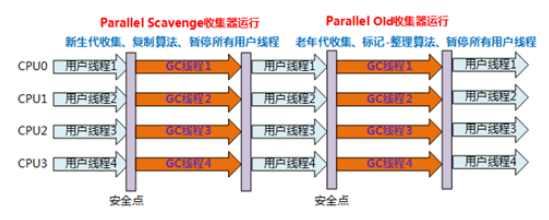
**1、特点**

      针对老年代；

      采用"标记-整理"算法；

      多线程收集；

Parallel Scavenge/Parallel Old收集器运行示意图如下：



2.应用场景

JDK1.6及之后用来代替老年代的Serial Old收集器；特别是在Server模式，多CPU的情况下；这样在注重吞吐量以及CPU资源敏感的场景，就有了Parallel Scavenge+Parallel Old收集器的"给力"应用组合。

3、设置参数

      "-XX:+UseParallelOldGC"：指定使用Parallel Old收集器。

### 5.6 CMS收集器

并发标记清理（Concurrent Mark Sweep，CMS）收集器，是以获得**最短回收停顿时间为**目标的收集器。基于“标记-清除”算法实现的。JDK1.5版本发布的。

1.CMS收集器运作过程

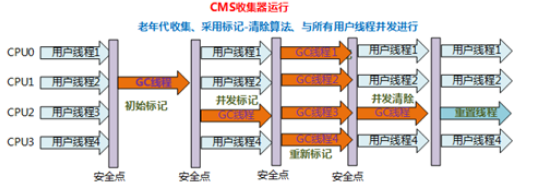
可以分为4个步骤:

（1）初始标记（STW）：仅仅记录GC Roots能直接关联到的对象，速度很快；

（2）并发标记：GC Roots Tracing的过程；（并发执行）

（3）重新标记（STW）：为了修正并发标记期间因程序的继续运作而导致标记产生表懂得那一部分对象的标记记录；

（4）并发清除。（并发执行）



2.优点

并发收集，低停顿

3.缺点

1.**对CPU资源非常敏感**。

在并发阶段，虽然不会导致用户线程的停顿，但是会因为占用了一部分CPU的资源而导致应用程序变慢。

2.**无法处理浮动垃圾**。

浮动垃圾即是在并发清除的过程中，伴随程序的运行自然就会有新的垃圾产生，这一部分垃圾CMS无法在当次收集中处理掉。 这使得并发清除时需要预留一定的内存空间，不能像其他收集器在老年代几乎填满再进行收集；

参数"-XX:CMSInitiatingOccupancyFraction"：设置CMS预留内存空间， JDK1.5默认值为68%；JDK1.6变为大约92%；

**"Concurrent Mode Failure"失败**

      如果CMS预留内存空间无法满足程序需要，就会出现一次"Concurrent Mode Failure"失败；

      这时JVM启用后备预案：临时启用Serail Old收集器，而导致另一次Full GC的产生；

      这样的代价是很大的，所以CMSInitiatingOccupancyFraction不能设置得太大。

**3. 产生大量内存碎片**

  由于**CMS基于"标记-清除"算法，清除后不进行压缩操作**。产生大量不连续的内存碎片会导致分配大内存对象时，无法找到足够的连续内存，从而需要提前触发另一次Full GC动作。

 解决方法：

（1）"-XX:+UseCMSCompactAtFullCollection"

      使得CMS出现上面这种情况时不进行Full GC，而开启内存碎片的合并整理过程；

      但合并整理过程无法并发，停顿时间会变长；默认开启（但不会进行，结合下面的CMSFullGCsBeforeCompaction）；

（2）"-XX:+CMSFullGCsBeforeCompaction"

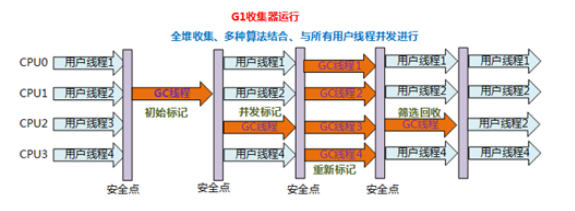
      设置执行多少次不压缩的Full GC后，来一次压缩整理；

      默认为0，也就是说每次都执行Full GC，不会进行压缩整理；

### 5.7 G1收集器

G1（**Garbage-First**）是**JDK7-u4**才推出商用的收集器。

**G1收集器运行示意图如下：**



### 1.特点

与其他收集器相比，GMS具有一下特点：

（1）并行与并发：  能充分利用多CPU、多核环境下的硬件优势；可以并行来缩短"Stop The World"停顿时间； 也可以并发让垃圾收集与用户程序同时进行；

（2）分代收集：能独立管理整个GC堆（新生代和老年代），而不需要与其他收集器搭配；

 能够采用不同方式处理不同时期的对象；

（3）空间整合：从整体看是基于“标记-整理”，从局部（两个Region间）看，是基于复制算法；这两种算法都不会产生空间碎片。

（4）可预测停顿：G1除了追求低停顿处，还能建立可预测的停顿时间模型；可以明确指定M毫秒时间片内，垃圾收集消耗的时间不超过N毫秒；

### 2. G1的分代收集

能独立管理整个GC堆（新生代和老年代），而不需要与其他收集器搭配。 能够采用不同方式处理不同时期的对象，虽然保留分代概念，但Java堆的内存布局有很大差别：

      将整个堆划分为多个大小相等的独立区域（Region）；

      新生代和老年代不再是物理隔离，它们都是一部分Region（不需要连续）的集合；

### 3. 为什么G1收集器可以实现可预测的停顿

  因为可以有计划地避免在Java堆的进行全区域的垃圾收集。G1跟踪各个Region获得其收集价值大小（回收所获得的空间大小以及回收所需的时间的经验值），在后台维护一个优先列表；每次根据允许的收集时间，优先回收价值最大的Region（名称Garbage-First的由来），这就保证了在有限的时间内可以获取尽可能高的收集效率。

### 4.G1收集器的运作步骤

（1）初始标记（STW）：仅标记一下GC Roots能直接关联到的对象；且修改TAMS（Next Top at Mark Start）,让下一阶段并发运行时，用户程序能在正确可用的Region中创建新对象；

（2）并发标记：进行GC Roots Tracing的过程；

（3）最终标记（STW）：为了修正并发标记期间因用户程序继续运作而导致标记变动的那一部分对象的标记记录。

      上一阶段对象的变化记录在线程的Remembered Set Log；

      这里把Remembered Set Log合并到Remembered Set中；

（4）筛选回收：首先排序各个Region的回收价值和成本； 然后根据用户期望的GC停顿时间来制定回收计划；最后按计划回收一些价值高的Region中垃圾对象。

 回收时采用"复制"算法，从一个或多个Region复制存活对象到堆上的另一个空的Region，并且在此过程中压缩和释放内存。

## 说说STW

Java中Stop-The-World机制简称STW，是在执行垃圾收集算法时，java应用程序的其他所有线程都被挂起（除了垃圾收集器之外）。Java中一种全局暂停现象，全局停顿，所有Java代码停止，native代码可以执行，但不能与JVM交互；这些现象多半是由于gc引起。

## 新生代GC和老年代GC（Minor GC和 Full GC）

Minor GC：是指发生在新生代的垃圾收集动作，因为java的大多数对象都具备朝生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。

Full GC/Major GC：是指发生在老年代的垃圾收集动作，出现了Full GC，通常会伴有至少一次的Minor GC（但也并非绝对）。Full GC的速度一般要比Minor GC慢10倍以上。

## 对象什么时候会出现在老年代

1. 对象优先在Eden中分配，当Eden中没有足够的空间分配时会促发一次Minor GC。每次Minor GC结束后，Eden区会清空，因为它会把Eden中还依然存活的对象放到Survivor中，当Survivor中放不下时，则由分派担保进入老年代中。
2. 大对象直接进入老年代中。-XX:+PretenuerSizeThreshold 控制”大对象的“的大小。即当创建的对象大于这个临界值时，则该对象直接进入老年代。
3. 长期存活的对象将进入老年代。虚拟机对每个对象定义了一个对象年龄（Age）计数器。当年龄增加到一定的临界值时，就会晋升到老年代中，该临界值由参数：-XX:MaxTenuringThreshold来设置。如果对象在Eden出生并在第一次发生Minor GC时仍然存活，并且能够被Survivor中所容纳的话，则该对象会被移动到Survivor中，并且设Age=1；以后每经历一次Minor GC，该对象还存活的话会被移动到另一个Survivor区中，并且Age=Age+1。
4. 动态对象年龄判定：如上所示，虚拟机并不总是要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，如果在Survivor区中相同年龄（设年龄为age）的对象的所有大小之和超过Survivor空间的一半，年龄大于或等于该年龄（age）的对象就可以直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄。

## 为什么要进行分代回收?

JVM使用分代回收测试，是因为:**不同的对象，生命周期是不一样的**。因此不同生命周期的对象采用不同的收集方式。可以提高垃圾回收的效率。

java虚拟机对对象的存储划分为三个代，年轻代（Young Generation）、老年代（Old Generation）和持久代（Permanent  Generation）。

**持久代：**

主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大。

**年轻代：**

年轻代分三个区。一个Eden区，两个Survivor区(一般而言)（8：1：1）。

大多数情况下，所有新生成的对象首先都是放在年轻代(如果是过大的对象也可能会直接在年老代生成)。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

当Eden区没有足够的空间进行分配时，虚拟机会发生一次Minor GC。

1.Eden区对外提供堆内存。当Eden区快要满了，则进行Minor GC(新生代GC)，把存活对象放入SurvivorA区，清空Eden区；Eden区被清空后，继续对外提供堆内存；

2..当Eden区再次被填满，此时对Eden区和SurvivorA区同时进行Minor GC(新生代GC)，把存活对象放入SurvivorB区，此时同时清空Eden区和SurvivorA区；

3.Eden区继续对外提供堆内存，并重复上述过程，即在 Eden 区填满后，把Eden区和某个Survivor区的存活对象放到另一个Survivor区，如此下去。

4.当某个Survivor区被填满，且仍有对象未被复制完毕时，或者某些对象在反复Survive 15次左右时，则把这部分剩余对象放到老年代区域；当老年区也被填满时，进行Major GC（老年代GC），对老年代区域进行垃圾回收。

**老年代：**

熬过一次Minor GC后对象进入survivor区，熬过一次就年龄加一。到一定程度就进入到老年代。（默认为15岁）。可以通过参数MaxTenuringThreshold来设置。

大对象直接进入老年代。

动态对象年龄判定：如果在Survivor空间中相同的年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就直接进入老年代。无需等到MaxTenuringThreshold中要求的年龄。

## 空间分配担保

## 解决的问题：

新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代。

**避免频繁的Full GC。**

在发生minor gc之前，虚拟机会检测 : 老年代最大可用的连续空间>新生代all对象总空间？

  1、满足，minor gc是安全的，可以进行minor gc。

  2、不满足，虚拟机查看HandlePromotionFailure参数：

    （1）为true，允许担保失败，会继续检测老年代最大可用的连续空间>历次晋升到老年代对象的平均大小。若大于，将尝试进行一次minor gc，若失败，即Minor gc后存活的对象远远高于平均值，则重新进行一次full gc。

（2）为false，则不允许冒险，要进行full gc（对老年代进行gc）。

## 在java 7 和 java 8中GC的区别

永久代在JDK8中被完全的移除了。所以永久代的参数-XX:PermSize和-XX：MaxPermSize也被移除了。元数据取代了永久代。这样使得GC收集发生了变化。  
垃圾回收不会发生在永久代，如果永久代满了或着超过了临界值，会出发完全垃圾回收（full gc)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小堆避免full gc是非常重要的原因。

JDK1.8中已经完全移除了永久带。类的元数据存放在native堆中，这个空间被叫做：元数据区。默认情况下，类元数据的分配仅受限于可用的本地内存。我们可以使用新的MaxMetaspaceSize参数限定类元数据可用的本地内存的数量。它类似于MaxPermSize。当类元数据区使用量到达MetaspaceSize（32位机客户端模式12M，32位服务器模式16M,64位机会更大）的时候，会触发垃圾回收，然后回收掉无用的类加载器和class对象。MetaspaceSize的值设置的过大会延长垃圾回收时间。垃圾回收过后，引起下一次垃圾回收的类元数据空间的大小可能会变大。

## 新生代怎么划分？老年代怎么划分？

Java堆中是JVM管理的最大一块内存空间。主要存放对象实例。

在JAVA中堆被分为两块区域：新生代（young）、老年代（old）。

堆大小=新生代+老年代；（新生代占堆空间的1/3、老年代占堆空间2/3）

新生代又被分为了eden、from survivor、to survivor(8:1:1)；

新生代这样划分是为了更好的管理堆内存中的对象，方便GC算法---复制算法来进行垃圾回收。JVM每次只会使用eden和其中一块survivor来为对象服务，所以无论什么时候，都会有一块survivor空间，因此新生代实际可用空间只有90%。

老年代GC（major gc）----------指发生在老年代的垃圾回收动作，所采用是的标记--整理算法。

老年代几乎都是经过survivor熬过来的，它们是不会那么容易“死掉”，因此major gc不会想minor gc那样频繁。

## java内存模型，堆的结构是什么（堆的划分）

JVM 的 Heap 堆内存在物理上被划分为两部分：Young Gen, Old Gen。

堆大小=新生代+老年代；（新生代占堆空间的1/3、老年代占堆空间2/3）

新生代又被分为了eden、from survivor、to survivor(8:1:1)；

新生代这样划分是为了更好的管理堆内存中的对象，方便GC算法---复制算法来进行垃圾回收。JVM每次只会使用eden和其中一块survivor来为对象服务，所以无论什么时候，都会有一块survivor空间，因此新生代实际可用空间只有90%。

老年代GC（major gc）----------指发生在老年代的垃圾回收动作，所采用是的标记--整理算法。

注意：Perm Gen 不是 Heap 的一部分。

## Minor GC会STW吗

目前所有的新生代gc都是需要STW的。

Serial：单线程STW，复制算法；  
ParNew：多线程并行STW，复制算法；  
Parallel Scavange：多线程并行STW，吞吐量优先，复制算法；  
G1：多线程并发，可以精确控制STW时间，整理算法；

## 如果经常出现full GC怎么定位代码哪里出了问题 ？

要考虑到发生full GC的原因：

Full GC频繁发生，意味着你的内存分配机制存在问题，也许是内存泄露，有大量内存垃圾不断在老年代产生；也许是你的大对象（缓存）过多；也有可能是你的参数设置不好，minor GC清理不掉内存，导致每次minor GC都会触发Full GC；还有可能是你的老年代大小参数设置错误，老年代过小等等原因。

1.年轻代空间不足；  
2.per Gen（永久代）空间满；  
3.CMS GC时出现Handlepromotionfailed和concurrent mode failure；  
4.统计得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间等。

# 三、虚拟机类加载机制

## 类加载过程

类加载的全过程包括：加载、验证、准备、解析和初始化5个阶段。

### 1.1 加载

在加载阶段，虚拟机需要完成一下3件事：

* 通过一个类的全限定名类获取定义此类的二进制字节流（可从多种渠道获取如Jar，war，网络等）；
* 将这个字节流所代表的静态存储结构转换为方法区的运行时数据结构；
* 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口（HotSpot虚拟机将这个对象存储在方法区中，该对象将作为程序访问方法区中这些类型数据的外部接口）

### 1.2验证

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机的自身安全。

验证阶段会完成下面4个阶段的检验动作：

1、文件格式验证

第一阶段要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。

该验证阶段的主要目的是保证输入的字节流能正确的解析并存储于方法区之内，格式上符合描述一个java类型信息的要求。

2、元数据验证

对字节码描述的信息进行语义分析，以保证其描述的信息符合java语言规范的要求。例如这个类是否有父类 、这个类的父类是否继承了不允许被继承的类（被final修饰的类）等等。

3、字节码验证

通过数据流和控制流分析，确定程序的语义是否是合法的，符合逻辑的。例如保证跳转指令不会跳转到方法体之外的字节码指令上等等。这阶段是整个验证过程中最复杂的一个阶段。

4.符号引用验证

对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验， 这一阶段的校验发生在虚拟机将符号引用转换为直接引用的时候，这个动作发生在连接的第三阶段（解析阶段）中发生。

### 1.3准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段。这些变量所使用的内存都将在方法去中进行分配。

这个时候进行内存分配的只包括类变量（被static修饰的变量），并不包括实例变量，实例变量是在对象实例化时随对象一起分配在java堆中。

### 1.4解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号应用：符号引用是一组以符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用是能无歧义的定位到目标即可。

直接引用：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。

解析主要是针对类或接口、字段、类方法、接口方法等。

### 1.5初始化

初始化阶段才真正开始执行类中定义的java程序代码。初始化阶段是执行类构造器<clinit>()方法的过程。执行静态变量的赋值操作以及静态代码块,完成初始化。初始化过程保证了父类中定义的初始化优先于子类的初始化.但接口不需要执行父类的初始化。

## 类卸载发生在什么时候？

 当类被加载，连接和初始化后，它的生命周期就开始了。那什么时候开始卸载呢？

JVM中的Class只有满足以下三个条件，也就是该Class被卸载（unload）：

   - 该类所有的实例都已经被GC，也就是JVM中不存在该Class的任何实例。  
   - 加载该类的ClassLoader已经被GC。  
   - 该类的java.lang.Class 对象没有在任何地方被引用，如不能在任何地方通过反射访问该类的方法

由Java虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。Java虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的类的Class对象，因此这些Class对象始终是可触及的。

## 双亲委派模型

虚拟机设计团队把类加载阶段中的“通过类的一个全限定名来获取描述此类的二进制字节流”这个动作放到java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。

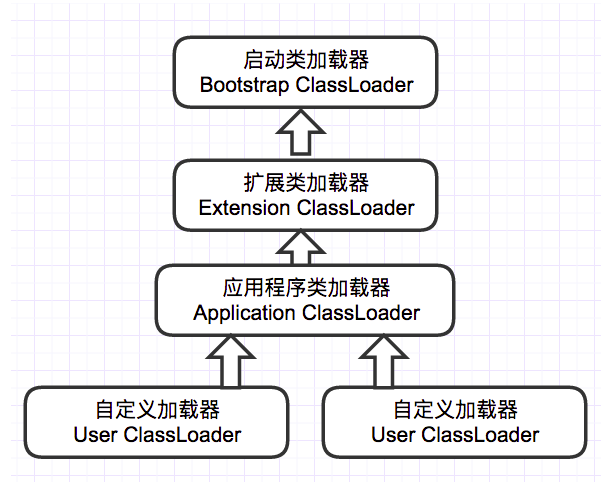
除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。顺序依次是:

Bootstrap ClassLoader: 启动类加载器，加载java\_home/lib中的类；

Extension ClassLoader: 扩展类加载器，加载java\_home/lib/ext目录下的类库；

Application ClassLoader: 应用程序类加载器，加载用户类路径上指定类库。

如果有必要，还可以加入自己定义的类加载器。这些类加载器之间的关系如图：



图中展示的类加载器之间的这种层次关系，称为类加载器的双亲委派模型。双亲委派模型要求除了顶层的启动类加载器之外，其余的类加载器都应当有自己的父类加载器。这里的类加载器之间的父子关系一般不会以继承的关系来实现，而是使用组合关系来复用父加载器的代码。

双亲委派模型的式作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完全这个加载请求时，子加载器才会尝试自己去加载。

使用双亲委派模型的好处在于**Java类随着它的类加载器一起具备了一种带有优先级的层次关系**。例如类java.lang.Object，它存在在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的Bootstrap ClassLoader进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，那系统中将会出现多个不同的Object类，程序将混乱。因此，如果开发者尝试编写一个与rt.jar类库中重名的Java类，可以正常编译，但是永远无法被加载运行。

1. 创建线程在内存模型中分配的位置
2. 熟悉内存模型，说一下内存划分？有哪些是线程安全的？
3. 虚拟机在加载类静态变量的时候是线程安全的吗，为什么、怎么实现的
4. 有做过jvm调优吗
5. 假设一个场景，要求 stop the world时间非常短，你会怎么设计垃圾回收机制？
6. 有没有用过JVM相关工具？
7. Java内存抖动严重，优化的思路
8. Java内存模型，原子性，可见性，有序性
9. 内存不够用时会回收软引用，那什么时候会发生这种事
10. 什么情况下会触发类加载 （）
11. string对象在JVM里面存储的位置